DTIC FILE COPY

(1)

219800-12-T

AD-A227 619

STAGECODE GENERATION WITH C4PL PROCEDURES

A User's Guide to Cyto-HSS Stage Programming in the C4PL Language Environment

Paul A. Kortesoja

JULY 1990

Prepared for:
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-5800

Contract No. 42-3638

ERIM
P.O. Box 8618
Ann Arbor, MI 48107-8618

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
219800-12-T		

4. Title and Subtitle	5. Report Date
Stagecode Generation with C4PL Procedures: A User's Guide to Cyto-HSS Stage Programming in the C4PL Language Environment	July 1990
	6. Performing Organization Code: ERIM
7. Author(s): Paul A. Kortesoja	8. Performing Organization Report No.: 219800-12-T
9. Performing Organization Name and Address: Environmental Research Institute of Michigan, P.O. Box 8618, Ann Arbor, MI 48107-8618	10. Work Unit No.: PTD-90-063
	11. Contract or Grant No.: 42-3638
	13. Type of Report and Period Covered: Programming Manual, Sept. 1989 - Oct. 1990
12. Sponsoring Agency Name and Address: Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-5800	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This document describes facilities within ERIM's C4PL image processing programming language that allow direct manipulation and generation of programming code for the Cyto-HSS Neighborhood Processing Stage. The Cyto-HSS Stage performs complex cellular (neighborhood) and point transformations on eight-bit images.

Since all feasible operations that can be performed by the stage cannot be produced with single commands within C4PL, methods have been provided to access the lowest level of programming of the stage. Algorithmists who need to transform images, in ways not directly supported by commands in C4PL, have the ability to set up their own low-level stage program data blocks. This provides the knowledgeable C4PL programmer with access to all the flexibility inherent in the stage hardware.

Stage code generation is an art that deserves to be hidden from the user--this is one of the reasons for the existence of C4PL. The Cyto-HSS Stage, like any other piece of specialized programmable digital hardware, is a relatively complicated unit. Detailed knowledge of its internal structure and operational characteristics is necessary in order to program effectively. However, the complicated nature of the beast is the source of its power; in the hands of knowledgeable and creative programmers, it can be made to perform wonderful deeds on digital data. This document is intended to help provide that necessary knowledge.

17. Key Words	18. Distribution Statement
Cellular Automata Computer Languages Image Processing Programming Languages	Approved for public release; distribution is unlimited.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	57	

CONTENTS

CONTENTS (concluded)

FIGURES

1.0 INTRODUCTION

This document describes facilities within ERIM's C4PL image processing programming language that allow direct manipulation and generation of programming code for the Cyto-HSS Neighborhood Processing Stage. The Cyto-HSS Stage performs complex cellular (neighborhood) and point transformations on eight-bit images.

C4PL is a powerful, general purpose language for constructing image processing algorithms to enhance digital images, and to extract useful data from digital images. Its library of commands that provide complex image transform operations with single-line commands covers a wide range of useful transforms. But it is nearly impossible, and not even desirable, to make all possible operations available within the Cyto-HSS stage accessible to the programmer as single-line commands.

Since all feasible operations that can be performed by the stage cannot be produced with single commands within C4PL, methods have been provided to access the lowest level of programming of the stage. Algorithmists who need to transform images in ways not directly supported by commands in C4PL have the ability to set up their own low-level stage program data blocks. This provides the knowledgeable C4PL programmer with access to all the flexibility inherent in the stage hardware.

Early versions of the C4PL language, and its predecessor C3PL, did not have the commands and capabilities necessary to directly program the stage from the language itself. External tasks were devised to provide this access to the lowest levels of the software system to allow generation of user-defined stage code blocks. This method of programming the stage is still available, but everything that in the past had to be done in external tasks can now be done at the C4PL command language level.

Stage code generation is an art that deserves to be hidden from the user--this is one of the reasons for the existance of C4PL. The Cyto-HSS Stage, like any other piece of specialized programmable digital hardware, is a relatively complicated unit that requires detailed knowledge of its interal structure and operational characteristics to program effectively. However, the complicated nature of the beast is the source of its power; in the hands of knowledgeable and creative programmers it can be made to perform wonderful deeds on digital data. This document is intended to provide the necessary knowledge.

1.1 REFERENCES

Several related documents and textbooks should be available for the stage programmer's reference.

The Stage Programmer's Manual provides a hardware-level description of the stage's internal operation. It also defines the organization and contents of the stage code program data block. (This is essential for C4PL programmers setting up their own stage code blocks.)

The C4PL Advanced Programming Manual provides much detailed information on writing external tasks in C4PL. External tasks (among other things) were provided in C4PL to create user-defined stage code blocks. As C4PL has evolved, however, the capabilities needed to create user-defined stage code have been developed within C4PL to the point where external tasks are no longer needed for this purpose. Useful information is contained in Chapters 3 (External Tasks in C), 4 (Operation and Programming the Cyto-HSS Stage), and 6 (Writing External Tasks in Pascal).

An introduction to C4PL and descriptions of basic capabilities and functions are contained in the C4PL User's Manual. From a stage code block development viewpoint, the interesting portions of the User's Manual are sections 4.2 through 4.4 (variables, constants, and expressions and operators). Understanding and knowledge of Chapter 5 material on C4PL procedures is essential for development of procedures to generate stage code blocks. Also, Chapter 11 is useful, as it describes saving stagecode and executing stagecode.

A number of textbooks exist that provide background information on relevant image processing and computer architecture concepts. For background information and examples of the concepts of cellular transformations see Cellular Automata by E.F. Codd. Further illustrations of cellular automata concepts may be found in Modern Cellular Automata by K. Preston and M.J.B. Duff. The theoretical foundations of mathematical morphology and its applications to image analysis are described in Image Analysis and Mathematical Morphology by J. Serra.

2.0 CYTO-HSS CONCEPTS

The Cyto-HSS Stage transforms pixels of an image in a programmably-defined way. An image is presented pixel-by-pixel to each stage in raster scan order from the first line through the last line. The stage transforms each pixel by evaluating the programmed transform of the 3 by 3 neighborhood of pixels around and including the input pixel. The stage retains the original pixel values for the two most recent lines of an image as it passes through, in order to have the data available for the 3 by 3 neighborhoods.

The "pipeline" in the Cyto-HSS is a serially-linked set of Cyto-HSS stages. Images passed through the pipeline will be transformed in a programmably-defined way in each stage. The resultant image output of each stage is passed to the input of the subsequent stage. In this way, multiple operations on an image can be performed in one "circulation" of an input image through the pipeline. The Cyto-HSS's power results from each stage's ability to produce a neighborhood transformed image pixel in one "clock tick" of the machine, and the ability to cascade stages together to multiply the number of neighborhood transforms that occur with each clock tick. The Cyto-HSS has as its primary purpose the support of high-speed circulation of data through this pipeline of stages (and other processing modules).

3.0 C4PL IMAGE PROCESSING COMMANDS THAT PRODUCE STAGECODE

C4PL provides dozens of commands to create image transformation operations. All of these commands are parameterized to the fullest extent possible to provide the maximum flexibility to the user for specifying the desired transformation.

Before proceeding with reading the detailed explanations of the "inner workings" of C4PL stagecode that constitute the rest of this document, an overview of the broad range of predefined image transformation commands available in C4PL that utilize the Cyto-HSS Stage will be given. This material is a large subset of the introductory material in the EPICAL Reference Manual.

Image transformations are operations which perform image processing transformation for image analysis and other purposes. These programs make up the EPICAL Library of C4PL. The types of transformations available in EPICAL include:

- Combining Images
- Geometric Transformations
- Global Transformations
- Local Transformations
- Point Transformations
- Translation-Based Operations

Local Transformations and Point Transformations (which encompass the large majority of the commands defined in C4PL) are the commands which utilize the Cyto-HSS stage. These families of commands are briefly outlined below. (For detailed information on any command, reference the C4PL EPICAL Reference Manual.)

3.1 LOCAL TRANSFORMATIONS OVERVIEW

Local Transformations are image processing operations that use the value of the neighboring pixels to determine the new value of each pixel.

The types of local transformations of the EPICAL Library include:

- Cellular Transformations
- Edge Detection
- Filters
- Maxima/Minima
- Morphological
- Shading

3.1.1 Cellular Transformations Overview

Cellular Transforms are image processing operations that perform neighborhood tranformations. Cellular transforms in the EPICAL Library include:

Ave4	Calculates the average of a subset of the 3 by 3 window (east, west, north and south), not including the center
Ave5	Calculates the average of a subset of the 3 by 3 window and then averages this value with the value of the original center
Ave8	Calculates the average of all eight neighbors, not including the center
Ave9	Calculates the average of the neighbors of the 3 by 3 window and then averages this value with the value of the original center
Aver	Replaces each pixel with the average of a specified set of its neighbors
Clampcen	Donut filter primitive. Eliminates discontinuities in the image by chopping off peaks and filling in negative-going valleys
Convolve	Performs a convolution using a 3 by 3 kernel
Countnei	Changes the state of each active pixel to the total of its active neighbors, not including the center
Countwin	Changes the state of each active pixel to the total number active neighbors, including the center
Findends	Finds endpoints of lines in specified state
Findtees	Finds T-connections in a rectangular skeleton, assuming 4-way connectivity
Markends	Marks endpoints of lines in specified state
Marktees	Marks T-connections in a rectangular skeleton, assuming 4-way connectivity
Match	Transforms all pixels whose neighbors match the specified pattern
Peakdete	Detects the peaks (i.e., local maxima)
Peelhex	Performs a series of hexagonal 2-D erosions on all objects in a specific state, modifying the state of the main regions and leaving the periphery in the original state
Shift	Translates an image by a specified distance in the specified compass direction (N, NE, E, SE, S, SW, W or NW).
Span	Conditionally dilates pixels in an image
Spandisk	Conditional 2-D dilation by a disk
Spanduod	Conditional 2-D dilation by a duodecagon
Spanv	Conditionally dilates pixels in an image over selected neighbors
Tran	Conditionally transforms the pixels in an image

Tranb	Conditionally transforms the pixels in an image using all selected neighbors
Tranbx	Conditionally transforms the pixels in an image using exactly the selected neighbors
Translat	Translates the active image X pixels to the east, and Y pixels to the south
Tranv	Conditionally transforms the pixels in an image using selected neighbors
See also:	Edge detection/gradient extraction, filters, maxima/minima, and morphological transforms

3.1.2 Edge Detection/Gradient Extraction Overview

The Edge Detection/Gradient Extraction routines preserve and/or enhance the regions of the image with local discontinuities. There are a very large number of techniques of this type, and each has different characteristics depending on the nature of the image, the objects of interest, and any noise or distortions present. Several different routines have been included in EPICAL, and more are being added as they evolve.

The following edge detection/gradient extraction operations are available in EPICAL:

Diff1	Takes the directional first difference of an image in the specified direction
Diff2	Takes the directional second difference of an image in the specified direction
Getedge4	Maximum of the local maxima of directional gradients in the north-south and east-west directions only, in rectangular coordinates
Getedges	Maximum of the local maxima of all directional gradients in rectangular coordinates
GradEW	Gradient in the east-west direction
Gradient	Maximum of all directional gradients in rectangular coordinates
GradNESW	Gradient in the northeast-southwest direction
GradNS	Gradient in the north-south direction
GradNWSE	Gradient in the northwest-southeast direction
Grad4	Maximum of north-south & east-west gradients
SlopeEW	Synonym for Diff1xx
SlopeNS	Synonym for Diff1yy
Sobel	Performs a Sobel edge detection using a 3 by 3 neighborhood on a rectangular image
Sobeldir	Computes the Sobel edge direction values

3.1.3 Filters

Filters are used to remove noise, such as details and distortions in the image outside the size range of interest which can cause difficulties and inaccuracies in processing the image. The filters listed below remove both light and dark (foreground and background) noise. To remove only one or the other type of noise use an opening or a closing. Many of the filters are iterative, progressively operating on the image with sequentially larger versions of the specified structuring element. This has the effect of removing larger and larger noise features.

2-D/Binary Filters

2D filters take a binary (or multi-state) image as input. The specified state is filtered, and the pixels which are changed can be put into another specific state. These filters will remove small regions, fill in small holes in blobs, and smooth the outlines of regions.

These filters can also be used to extract the small details. After filtering, the pixels which are different from the original image can be extracted (e.g., using an image subtraction) and used in subsequent algorithm steps.

3-D/Greyscale Filters

These filters treat the image data as a continuous sequence of increasing values. They treat the two-dimensional array of eight-bit pixels (the image) as a three-dimensional surface, with the value of each pixel representing the height of the surface at that point. In reality, the pixel value may represent intensity, range, color, or any other type of data value. These operations perform a neighborhood transformation over a three-dimensional neighborhood in rectangular coordinates. There is currently no software support for a three-dimensional hexagonal (footprint) neighborhood.

3-D/greyscale filters can be used for background normalization. Background normalization refers to a method of solving a common problem with grey-level images. The problem occurs when objects of differing depths/brightnesses need to be recognized on a varying background. Simple thresholding would not work because different objects may not have the same threshold, and the background itself may contain values above the threshold. The basic idea is to remove the objects from the background. This resultant image is then subtracted from the original image, causing the background to be removed. The objects are then readily discernable from the new background.

The following 2D/binary filters are available in EPICAL:

DiskFil   Disk filter-synonym for IsoFil2D

Hullfil	Performs an iterative filtering of an image by successively taking the convex hull of the foreground and background
IsoFil2D	2D Iterative isotropic filter

The following 3D/greyscale filters are available in EPICAL:

ArchFiEW	Iterative filter using an arch oriented in the east-west direction
ArchFiNS	Iterative filter using an arch oriented in the north-south direction
AutoMedian	Pseudo-median filtering
ClampCen	Clamp center (donut filter primitive)
ConeFil	Iterative filter using a cone
ConeTipF	Iterative filter using a cone with the origin at the tip
CubeFil*	Iterative filter using a cube
CylFil*	Iterative filter using an upright cylinder
DonutFil*	Iterative filter using a donut (ring shaped) structuring element
Donut1Fi*	First order donut filter (seven of eight neighbors)
Filterby	Iterative filter with specified structuring element
Gaussian	Convolves image with Gaussian kernel of specified size
HoleFil*	Remove (fill in) holes (isolated dark pixels)
IsoFil3D	3-D isotropic filter-synonym for SphereFi
Median	Replaces center pixel values with the median of the neighborhood values
PyramidF	Iterative filter using a pyramid
SphereFi*	Iterative filter using a sphere
SpikeFil*	Filter (remove) spikes (isolated bright pixels)
WallFiEW*	Iterative filter using a wall oriented in the east-west direction
WallFiNS*	Iterative filter using a wall oriented in the north-south direction

*Can also be used on binary images (but not multi-state images).

See also: Openings, Closings

3.1.4 Maxima And Minima Transformations Overview

The following Maxima and Minima operations are available in EPICAL:

LMax	Local Maximum--replaces center pixel with the maximum of the specified neighbors

LMin	Local Minimum--replaces center pixel with the minimum of the specified neighbors
Max3D	Replaces each pixel with the maximum of the neighbors value
MaxEW	Retains only those pixels which are a maximum with respect to their neighbors in the east-west direction
Maxex0	Replace the center pixel with the unbiased maximum of the neighborhood pixels except, when the center is zero
MaxNESW	Retains only those pixels which are a maximum with respect to their neighbors in the northeast-southwest direction
MaxNS	Retains only those pixels which are a maximum with respect to their neighbors in the north-south direction
MaxNWSE	Retains only those pixels which are a maximum with respect to their neighbors in the northwest-southeast direction
Maxov0	Replace the center pixel with the unbiased maximum of the neighborhood pixels, only if the center is zero
Min3D	Replaces each pixel with the minimum of the neighbors values
MinEW	Retains only those pixels which are a minimum with respect to their neighbors in the east-west direction
Minex0	Replace the center pixel with the unbiased minimum of the neighborhood pixels, except when the center is zero
MinNESW	Retains only those pixels which are a minimum with respect to their neighbors in the northeast-southwest direction
MinNS	Retains only those pixels which are a minimum with respect to their neighbors in the north-south direction
MinNWSE	Retains only those pixels which are a minimum with respect to their neighbors in the northwest-southeast direction
PeakDete	Retains only those pixels which are a peak with respect to their neighbors

3.1.5 Morphological Transformations Overview

Morphological Transformations are local image processing transformations based on geometric operations for image enhancements and shape analysis. The types of morphological transformations in the EPICAL library include:

2-D/Binary Transforms

3-D/Greyscale Transforms
Closings
Convex Hulls
Conditional Dilations
Dilations
Erosions
Openings
Skeletons
Size Encoding

See also: Cellular Transforms, Filters, Closings, and Openings

3.1.5.1 Binary(2-D) Transformations Overview

Two-dimensional image-processing commands are used to alter a digital image, taking into account the values of pixels which are adjacent to each other in the image.

SPAN and TRAN commands.

Span and Tran are the oldest 2-D image processing commands and have been superseded by the more general MATCH command. Though these two commands take the same parameters, the parameters have different names reflecting the conceptual difference in the commands. A span (dilate) command conceptually takes pixels in the source state and grows outward from the source over pixels in the medium state, changing medium state pixels to the resultant wave state. The tran (transform) commands conceptualize the transformation the other way around: if the pixel in the specified center state is surrounded by the specified configuration(s) of pixels in the neighbor state, then the center pixel is changed to the output state. (Note than any span command can be changed to a corresponding tran command by specifying neighbor = source, center = medium, output = wave, and reflecting the neighborhood specification (if any) across the center).

HEXFLG may be reset whenever a SPAN or TRAN command is given. To set it to true (i.e. use the hexagonal mode), append an H to the command (SPANH, TRANH). To set it to false, (i.e. use the rectangular mode), append an R to the command (SPANR, TRANR).

A special transformation can be enabled by giving a parameter value of "ALL" (meaning over any center or medium). The center or medium is transformed to the output state whenever the transformation test (neighbors of the specified value in the specified configuration) succeeds, regardless of original center or medium value.

See also: Cellular Transformations.

3.1.5.2 Greyscale(3D) Transformations Overview

A group of commands is included in C4PL which process image data in a 3-D manner. These commands treat the two-dimensional array of eight-bit pixels (the image) as a three-dimensional surface, with the value of each pixel representing the height of the surface at that point. In reality, the pixel value may represent intensity, range, color, or any other type of data value. These operations perform a neighborhood transformation over a three-dimensional neighborhood in rectangular coordinates. There is currently no software support for a three-dimensional hexagonal (footprint) neighborhood.

See also:

Closings
Dilations
Erosions
Filters
Openings

3.1.5.3 Closings Overview

A Closing is a dilation followed by an erosion with the same structuring element. Closings remove isolated dark points, concavities and background regions smaller than the structuring element which is used. The following closings by structuring elements are available:

ClArchEW	Closing by an arch oriented in the east-west direction
ClArchNS	Closing by an arch oriented in the north-south direction
ClConeTi	Closing by of a cone with the origin at the tip
CloseCon	Closing by a cone
CloseCub	Closing by a cube
CloseCyl	Closing by an upright cylinder
CloseDis	Closing by a disk
ClosePyr	Closing by a pyramid
CloseSph	Closing by a sphere
ClWallEW	Closing by a wall oriented in the east-west direction
ClWallNS	Closing by a wall oriented in the north-south direction
Proper_Closing	Filtering operation to remove localized dark features

See also: Openings and Filters.

3.1.5.4 Convex Hulls Overview

A convex hull is the smallest convex shape which contains the figure. This is roughly equivalent to placing a rubber band around each connected region of foreground pixels, changing the pixels which are in the concavities and holes. The convex hull routines in EPICAL compute approximations limited by the digital grid space. Increasing the number of sides used improves the accuracy, but increases the execution time. The following convex hull transformations are available in EPICAL on two-dimensional (binary or multi-state) images:

Hull12	Duodecagonal (twelve sided) hull (hexagonal grid)
Hull16	Sixteen-sided hull (rectangular grid)
Hull4	Rectangular hull (rectangular grid)
Hull6	Hexagonal hull (hexagonal grid)
Hull8	Octagonal hull (rectangular grid)

3.1.5.5 Conditional Dilations Overview

The following Conditional Dilations are available in the EPICAL Library:

Match	Transforms all pixels whose neighbors match the specified pattern
Span	Conditionally dilates pixels in an image
SpanDisk	Span Disk (Rectangular)
SpanDuod	Span Duodecagonal (Hexagonal)
Tran	Conditionally transforms the pixels in an image

3.1.5.6 Dilations Overview

The following Dilations by 3-D structuring elements are available in the EPICAL Library:

DArchEW	Dilate by an arch oriented in the east-west direction
DArchNS	Dilate by an arch oriented in the north-south direction
DCone	Dilate by a cone
DConeTip	Dilate by a cone with the origin at the tip
DCube*	Dilate by a cube
DCyl*	Dilate by an upright cylinder
Dilate	Expands an image, treating it as a three-dimensional surface
DPyramid	Dilate by a pyramid
DSphere	Dilate by a sphere
DwallEW*	Dilate by a wall oriented in the east-west

	direction
DwallNS*	Dilate by a wall oriented in the north-south direction
Lmax*	Replaces the center pixel with the maximum of the enabled neighborhood pixels
Match	Transforms all pixels whose neighbors match the specified pattern
Span	Conditionally dilates pixels in an image
Spandisk	Conditional 2-D dilation by a disk
Tran	Conditionally transforms the pixels in an image
2-D Dilations	Dilations on 2-D binary/multistate images

*Can also be used on binary (but not multistate) images

3.1.5.7 Erosions Overview

Erosions by the following 3-D structuring elements are available for use in the EPICAL library:

EArchEW	Erode by an arch oriented in the east-west direction
EArchNS	Erode by an arch oriented in the north-south direction
ECone	Erode by a cone
EConeTip	Erode by a cone with the origin at the tip
ECube*	Erode by a cube
ECyl*	Erode by an upright cylinder
EPyramid	Erode by a pyramid
Erode	Shrinks an image, treating it as a three-dimensional surface
ESphere	Erode by a sphere
EwallEW*	Erode by a wall oriented in the east-west direction
EwallNS*	Erode by a wall oriented in the north-south direction
Lmin*	Replaces the center pixel with the minimum of the enabled neighborhood pixels
Match	Transforms all pixels whose neighbors match the specified pattern
Span	Conditionally dilates pixels in an image
Spandisk	Conditional 2-D dilation by a disk
Spanduod	Conditional; 2-D dilation by a duodecagon
Tran	Conditionally transforms the pixels in an image
2D Erosions	Erosions on 2-D binary/multistate images

*Can also be used on 2-D binary (but not multistate) images

3.1.5.8 Openings Overview

Opening is an erosion followed by a dilation with the same structuring element. Openings remove isolated bright points, convexities and foreground regions smaller than the structuring element which is used. The following openings by structuring elements are available:

OpArchEW	Opening by an arch oriented in the east-west direction
OpArchNS	Opening by an arch oriented in the north-south direction
OpConeTi	Opening by a cone with the origin at the tip
OpenCone	Opening by a cone
OpenCube*	Opening by a cube
OpenCyli*	Opening by an upright cylinder
OpenDisk	Opening by a disk (2-D images only)
OpenPyra	Opening by a pyramid
OpenSphe	Opening by a sphere
OpWallEW*	Opening by a wall oriented in the east-west direction
OpWallNS*	Opening by a wall oriented in the north-south direction
Proper_Opening	Filtering operation to remove localized bright features

*Can also be used on 2-D binary (but not multistate) images

3.1.5.9 Skeletons Overview

A Skeleton is stick figure that results when a region is thinned with a connectivity preserving algorithm. Mathematically speaking, it is all of the pixels which are equidistant from two or more background pixels. The skeletonizing routines in EPICAL remove pixels on the perimeter of the foreground regions if they are not on the skeleton. These routines work from only one direction on each step, proceeding sequentially around the regions so that thin lines will not be broken. Skeleton procedures available in EPICAL are:

SkelHex	Produces a skeleton with hex connectivity (hexagonal grid)
SkelRec4	Produces a skeleton with N, S, E, and W connectivity (rectangular grid)
SkelRec8	Produces a skeleton with eight neighbor connectivity (rectangular grid)

In addition, the following routines are available which thin the foreground regions from only a single direction:

ReduceE	Reduce from the east side
ReduceN	Reduce from the north side
ReduceS	Reduce from the south side
ReduceW	Reduce from the west side

3.1.5.10 Size Encoding Overview

Size Encoding is the labeling of each foreground pixel with a value representing its distance to the nearest background regions. Functions of this class are called distance transforms. The following size encoding routines are available in EPICAL:

SizeEncdR	Size encoded erosion of an image by a 45-degree cone (pyramid) whose origin is at its tip (rectangular grid)
PeelHex	Performs a series of hexagonal 2-D erosions on all objects in a specified state, modifying the state of the main regions and leaving the periphery in the original state on each pass

3.1.6 Shading Overview

Shading operations treat a grey-scale image as a three-dimensional surface and selectively lighten or darken the image to provide the appearance of depth due to directional illumination. The following shadings are available in EPICAL:

Shade	Shade a grey-scale image as if there were a light source in the upper right hand corner of the screen
Shadow	Shadow a grey-scale image as if there were a light source in the upper right hand corner of the screen, given the length of the shadow to cast

See also: Plot3D

3.2 POINT TRANSFORMATIONS OVERVIEW

Point Transformations are operations that take in one image and modify the pixel values based only on the values themselves; that is they ignore the neighborhood of pixels around them.

The pixel transformations performed by the commands in this category are actually carried out in the hardware of the Cyto-HSS. In each stage a 256 by 8 lookup table--the PRAM (Point-transform Random Access Memory)--is used. This table is loaded with the desired pixel values, and the original value serves as an index into this table.

When no transformation is desired, the table is bypassed. This means that these operations are carried out quickly and, in fact, take no additional time when done in concert with other operations performed by the stages. Any transformation is possible, and a number of useful ones have been included in EPICAL. The EPICAL commands that perform point transformations are briefly described below.

Abs	Takes the absolute value (i.e., values 128 to 255 are mapped into values of 127 to 1)
BitAnd	Logical ANDs between two bit planes
BitClear	Sets the bit plane to zero
BitClr	Synonym for Bitclear
BitCopy	Copies a bit plane
BitNot	Logical complement of a bit plane
BitOr	Logical OR between two bit planes
BitRot	Rotation (barrel shift) of the bits of each pixel
BitSet	Sets a bit plane to one
BitSwap	Exchanges two bit planes
BitXor	Logical exclusive-OR between two bit planes
Cover	Covers one pixel value with another
Exch	Exchanges two pixel values
Exp	Exponential function
Log2	Function returning log base 2 of an argument
LogE	Natural logarithm function
OnesComp	Ones complement of an image
Prune	Changes pixel values within a given range
Quant	Sets the pixels in a given range (or ranges) to a single value (or values)
Remove	Sets all values within a specified range to zero
Scale	Scales the pixels by multiplying, dividing or adding constants to the values
ScaleRem	Produces a remainder image consistent with SCALE
SetDR	Sets the dynamic range of an image by rescaling pixel values based on the range of actual values present
Slice	Segments an image into two states at a specified threshold level
SQRT	Square root function
SQR	Square function
Threshol	Sets all values below a specified level to zero
TwosComp	Twos complement of an image

4.0 WHAT IS "STAGECODE"?

Stagecode is the data that is loaded into a stage to configure its hardware calculation circuitry and load constant registers and lookup table RAMs. This data defines the transformation to be performed by the stage. At the hardware level, a stagecode block for one hardware stage consists of 790 or 798 bytes (eight-bit bytes) of data. The longer 798 byte data block is provided to program Cyto-HSS chip stages. All stage types (board and chip) support the 790 byte format data block.

C4PL abstracts the data block necessary to program the stage into two distinct types. These types are neighborhood transform operations and point transform operations. Although one hardware stage can be programmed to perform two transforms simultaneously (one of each type), C4PL (for reasons of logical clarity and system software considerations relating to the handling and optimization of sequences of stage operations) deals in stage code blocks of these two distinct types. These stage code types will be discussed in detail in subsequent sections.

At this point, we must clarify the terminology used in C4PL regarding stage code blocks. C4PL handles stage program information in a hierarchically structured way. At the lowest level are the actual data values that will ultimately be programmed into the stage hardware registers and lookup tables. Groupings of this data are formed to create the two distinct types of transformations possible in the stage. These data blocks are known as "stageops," or stage operations. Stageops, in turn, are grouped into sequences that can be identified uniquely in C4PL as "stagecode" and stored in C4PL variables. A stagecode variable in C4PL amounts to a list of identifiers, each of which identifies a stageop. A stageop can exist independently in a C4PL variable, although this is unusual. The hierarchy of stage program data organization is shown in Figure 1:

```
Stagecode:
---------
                            Stageops:
                 ---------------------------------
-------
| #ops|
-------                                    ---------
|op1id| ------------------------------> | optype|
-------                  ---------         ---------
|op2id| ---------> | optype|              |RAMtype|
-------                  ---------         ---------
                         |RAMtype|         |pgmdata|     | stage
                         ---------         ---  ---     | program
                         |pgmdata|         | etc...|     | data
                         ---  ---          ---------     |
                         | etc...|
                         ---------
```

FIGURE 1. Hierarchy of Stage Program Data Organization

Fundamentally, it is the stage program data in stageops that one creates and modifies to produce unique image transformations to be performed within the Cyto-HSS stage. However, the current context of stage data may vary depending on how it is created or manipulated, and this context must always be kept in mind. Stage data may be created directly and stored in C4PL array variables; or it may exist as a stageop variable where it is accessed by indexing; or it may only exist in the context of a stagecode variable, where it may still be accessed through indexing. Indexing into stageops must take into account the header information resident in the stageop variable.

Stage program data is always of C4PL type T_BYTE, that is, eight-bit unsigned integers. Calculations used to produce stage program data may use higher precision, but a scaling or truncation operation must be performed prior to placing such data in stagecode. For example, data may be created as T_INT (32-bit signed integer) arrays, then placed in stagecode through the ArrayToCode command. This command requires T_BYTE type data as input, so the T_INT array must be explicitly converted to T_BYTE prior to use of the ArrayToCode command. This conversion may be done with the MakeArray command.

The structure and content of stageops will now be defined further. Point transformation-type stageops will be discussed first, then neighborhood transformation-type stageops will be described. In the discussions that follow it is assumed that the reader has reviewed the <u>Stage Programmer's Manual</u> [1] to gain some familiarity with the operations that are performed within the stage.

4.1 POINT TRANSFORMATIONS

The PRAM (Point Random Access Memory) operation is a simple mapping of each pixel value to some new value. A input pixel's neighbors have no effect on this operation. A PRAM stageop data block consists of information about the contents of the 256-byte PRAM in the stage. A PRAM operation is expressed mathematically and graphically in Figure 2:

```
new_c = f(c)

 ------     f     ---------
 | c  |    -->    | new_c |
 ------           ---------
```

FIGURE 2. A PRAM Operation

Stageops of the point transform type are arrays of bytes in the order and with meanings as shown in Figure 3:

```
-------------------------------------------
|              optype (PRAM)              |
-------------------------------------------
|      RAMtype (RAW,ASCEND,DESCEND)       |
-------------------------------------------
| PRAM 0 (not present if RAMtype ~=RAW)   |
---                                     ---
|                 etc...                  |
---                                     ---
| PRAM 256 (not present if RAMtype ~=RAW) |
-------------------------------------------
```

FIGURE 3. Organization of PRAM Stageop Data

4.1.1 OpType

The first byte of all stageops defines the type of stageop. Point transformation operations have the predefined type PRAM.

4.1.2 RAMtype

The second byte defines the PRAM section of the subsequent data in the block. Three options are defined: RAW, ASCEND, and DESCEND. RAW means that all 256 locations of the PRAM are explicitly specified by 256 bytes of data that follow the RAMtype byte. ASCEND indicates that no data follows the RAMtype byte, and the PRAM is to be filled with data that defines a straight-through mapping (i.e., data=address). DESCEND indicates no data follows the RAMtype byte, and the PRAM is to be filled with an inverting mapping (i.e., data=NOT(address)).

4.1.3 PRAM Data

Either 0 or 256 bytes of data follow the RAMtype stageop parameter, depending on that parameter's value. This data will be programmed into the PRAM lookup table within the stage and defines the desired PRAM output value for each PRAM input value (a point-to-point mapping). This RAM consists of 256 bytes due to the 8-bit pixel resolution of the stage and data paths within the Cyto-HSS.

4.2 NEIGHBORHOOD OPERATIONS

A neighborhood transform operation consists of data for the stage control registers (of which there are 22 or 32 depending on the stage hardware to be programmed) and information relating to the contents of the 512-byte NRAM (Neighborhood Random Access Memory). Neighborhood transform operations are those operations that produce a new pixel value for each input value based on some function of the original pixel and one or more of its 8 neighbors. This function can be expressed in Figure 4:

```
new_c = f(ne,e,se,s,sw,w,nw,n,c)

----------------
| nw |  n | ne |
----------------     f     ----------
|  w | c  |  e |    -->    | new_c |
----------------           ----------
| sw |  s | se |
----------------
```

FIGURE 4. An NRAM Operation.

Stageops of the neighborhood transform type are arrays of bytes where each byte has meaning as shown in Figure 5:

```
---------------------------------------
|         optype (XFORM,XFORM2)       |
---------------------------------------
|  RAMtype (RAW,LNRAMO,HNRAMO,BOTHO)  |
---------------------------------------
|        control register 1           |
---                                 ---
|              etc...                 |
---                                 ---
|   control register N (N=22 or 32)   |
---------------------------------------
| NRAM 0 (not present if RAMtype=BOTHO)|
---                                 ---
|              etc...                 |
---                                 ---
|  NRAM N (not present, or N=256,512) |
---------------------------------------
```

FIGURE 5. Organization of NRAM Stageop Data.

4.2.1 OpType

The first byte of all stageops defines the type of stageop. Neighborhood operations may be one of two predefined types: XFORM or XFORM2. Type XFORM specifies a short code block containing 22 bytes of control register information. Type XFORM2 specifies the long program containing 32 bytes of control register bytes.

4.2.2 RAMtype

The second byte defines the NRAM section of the subsequent data in the block. Four options are defined: RAW, low-half NRAM all zeros(LNRAMO), high-half NRAM all zeros (HNRAMO), and both halves all zero (BOTHO). RAW means that all 512 locations of the NRAM are specified by 512 bytes of data that follow the control register data. LNRAMO indicates that 256 bytes of data follow the control registers and that these 256 bytes specify the high half of the NRAM (the low half is to be filled with all 0 data). Similarly, HNRAMO indicates that 256 bytes follow the control register data and that these 256 bytes specify the low half of the NRAM (the high half is to be filled with all 0 data). BOTHO indicates that no NRAM data follows the control registers, and that the entire NRAM is to be filled with zeros.

4.2.3 Control Registers

Control register data is organized in the order specified in the <u>Stage Programmer's Manual</u> [1]. A XFORM stageop will have 22 bytes of data and a XFORM2 stageop will have 32 byte positions. Figure 2 in the referenced document (page 9) shows the layout and meaning of the control register data for the 22-byte format. Figure 6 (page 26) shows the XFORM2 register format. Note that only 31 byte-wide control registers are defined in the long stage program format. The XFORM2 stageop data block definition allocates 32 bytes for the control registers for reasons relating to byte/word alignment. The extra byte is placed at the end of the 31 bytes of control register data and the value of this byte is of no importance. However, when indexing into XFORM2-type stageops, this dummy byte must be taken into account when calculating offsets to access NRAM data bytes (if present).

4.2.4 NRAM Data

Either 0, 256, or 512 bytes of data follow the control register data, depending on the value of the RAMtype paramter in the stageop definition. This data will be programmed into the NRAM lookup table in ascending address order within the stage, and defines the desired NRAM output value for each combination of neighbor pixel test condition evaluation results. The NRAM consists of 512 bytes because there are 9 neighbors in the 3 by 3 transformation window. The NRAM is accessed with a 9-bit address formed by the result (true/false) of the programmed evaluation condition (as specified by the control registers) for each of the nine neighbors. The bit position assignment within this address for each neighbor test result begins with the northeast neighbor at bit 0 (the low-order bit) and proceeds clockwise around the 3 by 3 window. The center pixel test result provides bit 8 (the high-order bit) of the NRAM address vector.

5.0 WHY MESS WITH STAGECODE DIRECTLY?

Many C4PL users will never need to understand nor manipulate stagecode directly. The C4PL language exists to provide an abstract, structured, and powerful "front-end" to the Cyto-HSS hardware. The language shields the user from having to be knowledgable about the intricacies of the hardware itself and from having to program image transformation operations at a language level that is unsuited to the operation being performed.

There may come a time, however, when the user wants to perform an operation that doesn't quite fit within the parametric constraints of the commands provided by C4PL. It is also possible to conceive a stage operation unlike anything that C4PL provides (this is an unusual situation given the breadth of commands and options within C4PL). These are the times when direct stagecode manipulation or generation may be required.

5.1 MODIFICATION OF STAGECODE

An example of a minor modification to an existing C4PL command follows. Suppose you want to find all 4-way connected cross points of pixels in state 5. A Match command specification to do this and save the resulting stagecode is as follows:

```
declare code              ; create a variable to hold stagecode

Match  2 ~5  5 ~5  &      ; create transformation, put in code
          5  5  5  &
         ~5  5 ~5       -> ,code
```

The code that results from this Match command is a XFORM-type stageop that would program all neighbor contribution values to equal -5, set the test condition for each neighbor for "equal 0", and put the output state (2) in appropriate locations in the high half of the NRAM. All other locations in the NRAM would be set to zero. The NRAM address vector generated in the neighborhood calculation logic in the stage would produce an NRAM output of either 0 or 2, depending on which neighbors pass the test. The output selection logic would pass the NRAM output on to the PRAM if it is non-zero, or the original center pixel value if the NRAM output is zero. The PRAM (although not specified in the stagecode that results from this Match command) would be set to an data=address or "straight" pattern to pass the output of the output selection logic through unaltered.

Now, suppose you want to mark all cross points having a center pixel state of 5 or greater. The Match command does not allow a pixel state specification of ">=5" or ">4", so the desired transform cannot be specified directly. But the stage can perform this desired operation. A "greater than or equal to 5" test on the center pixel value can be specified by setting the test condition for the center to check for a "carry" instead of an "equal to." Stagecode to perform this modified Match operation can be generated using the code that results from the Match command above and modifying it as follows:

```
        ; define symbol for 15th byte of stageop (center control)
syn c_control "15"

        ; neighbor test control bits = "01" (carry out)
syn carry_out "2"

        ; change center test to carry instead of equal zero
code[1][c_control] := code[1][c_control] | carry
```

This operation modifies the Test Control bits (TC1,TC0) in the neighbor control register for the center pixel to select the "carry" (0,1) condition instead of the "equal zero" condition (0,0). There is only one stageop in "code" (since it was declared just prior to the Match command), hence the first index is 1. The second index, or offset into the neighborhood transform stageop is 15. This value is required because the center pixel neighbor control register is the 13th byte of the stage program (see Figure 2, page 9 of the Stage Programmer's Manual [1]), and there are two "header" bytes at the begining of the stageop that define its type and RAM data content.

5.2 GENERATION OF STAGECODE

An example of an operation created by direct generation of stagecode follows. When no C4PL command exists that can produce stagecode similar to what is needed, then direct generation of stagecode in arrays is appropriate. Suppose a unique scaling operation was desired that would transform an image containing pixel values in the range of 0 to 255 to the range of 128 to 255. This could be done with a PRAM operation in a stage that maps each adjacent pair of input values to one output value (0 and 1 map to 128, 2 and 3 map to 129, ..., 254 and 255 map to 255). A PRAM stageop to do this transformation can easily be generated as follows:

```
declare pramarray,code,i              ; array,code,loop vars
makearray T_BYTE 256 -> pramarray     ; 256 byte array
for i 0 255                           ; fill with map data
  pramarray[i+1] := (i/2)+128
endfor
arraytocode pramarray -> ,code        ; convert to stagecode
```

In both of the examples above, the resulting stagecode in the variable "code" can be applied to images with the Apply command:

```
apply code 1 inputimage -> outputimage  ; execute stagecode "code" once
```

More complex examples of stagecode modification and generation are given later in this document.

6.0 C4PL FEATURES USEFUL FOR STAGECODE OPERATIONS

There are several commands and features of the C4PL language that allow and facilitate direct operations on and generation of stagecode. A brief description of each of these commands and features is presented below. For more detailed information, the C4PL User's Manual [2] or on-line help explanations may be referenced.

6.1 C4PL STAGECODE GENERATING COMMANDS

As seen in previous sections, there are dozens of commands available in C4PL that generate stagecode. For many applications, these commands provide more than enough power and flexibility to process images as required, without the user having to have any detailed knowledge of the stagecode that these commands produce.

Any C4PL or EPICAL command of the form

```
CommandName p1,p2...  inputimage -> outputimage, code
```

returns the stagecode it generates to perform the requested operation as an optional output. Stagecode is appended to variable "code" if it is specified. If "code" is not already a stagecode variable, it will be converted to one.

This command syntax for stagecode generating commands allows any of these commands to be utilized for generation of stagecode for subsequent customization by the user. The resulting code can be accessed directly by indexing into the stagecode variable, or the stageop or stageops contained in the stagecode variable can be converted into arrays for subsequent manipulations.

The Match command is particularly useful for creating a neighborhood transform stageop, and Cover is one command that may be used to produce a point transformation (PRAM-type) stageop.

6.2 ARRAYTOCODE COMMAND

This command allows the user to put data from C4PL arrays (of bytes) into a stagecode variable. The command does all the processing necessary to construct the appropriate stageop and attach it to the specified stagecode variable. Two different forms of the command exist, one for each type of stageop in C4PL:

```
ArrayToCode lNRAM hNRAM cntlregs inimage -> outimage, code
```

```
ArrayToCode PRAMarray inimage -> outimage, code
```

Note that this command has the same syntax as any other stagecode generating command in C4PL. The stage operation will be executed immediately with the specified images if the "code" variable is not specified. Otherwise, the stageop that results from the specified array data is appended to the "code" variable.

6.3 CODETOARRAY COMMAND

This command is the inverse operation to ArrayToCode. It also has two forms:

```
CodeToArray code opnumber -> lowNRAM highNRAM controlregs

CodeToArray code opnumber -> PRAMarray
```

The "opnumber" parameter specifies which stageop is to be converted from the list of stageops contained in "code". Although stageops may be manipulated directly via indexing, conversion to arrays may be useful in some cases to allow calculations on the stage program data to be done at higher precision.

6.4 APPLY COMMAND

Apply provides the means for executing stagecode once it has been generated and stored in a stagecode variable. The syntax is:

```
Apply stagecode passes inputimage -> outputimage
```

The same stagecode may be executed multiple times on the same inputimage by specifying a "passes" parameter greater than 1.

6.5 ARRAY OPERATIONS

Array operations and subarray notation in C4PL can provide efficient means of creating stage program data in arrays. Arrays in C4PL are similar to arrays in other programming languages. Whole arrays can be specified as operands for arithmetic, relational, and assignment operators. Array elements are referenced in the usual way via an array name with subscripts.

Subarray notation provides convenient access to subsets of arrays. It allows users to easily manipulate array information without the use of time-consuming loops and element-by-element indexing.

Stageops in C4PL can be treated as arrays. Images in C4PL can also be accessed and treated as arrays. These capabilities present a number of interesting possibilities, such as using image processing operations to create data in an image which can then be converted to and used as stagecode.

Array operations on stageops will only affect the stage data. The optype and RAMtype header bytes will be unaffected. For example, if "code" is a stagecode variable containing one PRAM type operation (that has a "RAW" or fully specified 256-byte data block), then that PRAM stageop may be directly set to all one value (e.g., 255) with the following subarray expression:

```
code[1][*] := 255
```

Please refer to the C4PL User's Manual section 4.2.3.1 "Arrays" for detailed information on arrays and subarray notation [2], or access the on-line help information.

6.6 STAGECODE ARRAY INDEXING

Array indexing of stagecode variables is possible in C4PL. This makes direct array operations on stageops contained in stagecode variables available to the user. Care must be taken to insure that the desired data is being accessed at all times, however.

Recall that a stagecode variable is a list of identifiers of stageops, which actually hold the stage program data. The stageop which has the data also has a two byte header in it to identify the type of operation and the type and amount of RAM data it contains. An indexed reference to a data location in a stageop contained in a stagecode variable must identify which stageop is being referenced as the first index, and the byte in the referenced stageop as the second index. For example, the first stage data byte of a XFORM type stageop which is the first stageop in a stage code variable "code" can be accessed as follows (note that in C4PL, all arrays are one-origin):

```
mode_control := code[1][3]
```

Note that 3 is specified as the second index. This locates the first stage data byte (the Mode Control register). The first two bytes of the XFORM stageop identify the stageop type, and the RAM type. The syntax is [m][n] because each index represents an index into a different one dimensional array. A syntax of [m,n] would represent a particular element of a single two dimensional array.

6.7 BIT-WISE LOGICAL OPERATORS

The relational operators & (logical AND), | (logical OR), and ~

(logical NOT) are very important for modifying particular bits in stage program data bytes. Several stage program bytes contain bits or bit subsets that control stage operation. These program bytes include Mode Control, the bit masks (Input and Output), and the nine Neighbor Control registers.

6.8 ROTATECODE COMMAND

RotateCode "rotates" the first stageop in a stagecode variable. The stageop must be a neighborhood transform type stageop (rotation of PRAM-type stageops has no logical meaning). Rotation in this context refers to the rotation of the neighbor program data about the center in the 3 by 3 neighborhood. Consider a stagecode variable produced with a Match command, such as:

```
Match 2  # ~1 ~1  &
         1  1 ~1  &
         # ~1 ~1  -> ,StageCode
```

A rotation of this StageCode by 90 degrees produces a different stagecode:

```
RotateCode StageCode 90 -> ,NewStageCode
```

that is equivalent to a Match command specified as follows:

```
Match 2  #  1  #  &
        ~1  1 ~1  &
        ~1 ~1 ~1  -> ,NewStageCode
```

Rotation of a stageop is accomplished by rotating the 8 neighbor control registers, the 8 neighbor contribution values, and rearranging the NRAM appropriately. In a 90-degree rotation the north-east neighbor control and contribution would take on the values previously contained in the north-west control and contribution registers. In all, each neighbor register value would shift two positions around the neighborhood in a clockwise direction. The NRAM rearrangement is somewhat complicated at first glance, but it is simply a rearrangement of data within the NRAM according to a rotation of the bits comprising the NRAM address vector. The bits of the 9-bit vector correspond to neighbors as follows:

C	N	NW	W	SW	S	SE	E	NE
8	7	6	5	4	3	2	1	0

Keep in mind that the NRAM is a 512-byte array with address values from 0 to 511, and therefore a rotation of 90 degrees maps the original data contained in each NRAM location to a new location specified by an address which is the original address with the low-order 8 bits rotated left (with wrap). For example, the NRAM data at address 1 (NE bit ON) would be placed in the new NRAM at address 4 (SE bit ON). Since the center does not rotate, each NRAM half is treated independently. Therefore, a 90 degree rotation would also take the data from address 257 (NE,C bits ON) and place it at new NRAM address 260 (SE,C bits ON), and so on.

6.9 ROTATEARRAY COMMAND

The RotateArray command provides the same capability to rotate stage program data as the RotateCode command; however, RotateArray operates on data stored as arrays. This command allows input arrays of 22 bytes (a standard stage control register block), 31 or 32 bytes (a long stage control register block), and 256 bytes (an NRAM half). The syntax is the same as the RotateCode command.

6.10 ASCENDING_PRAM

At some point in the near future, C4PL will have a system constant 256-byte array containing data whose value equals the array index (an "ascending" PRAM). This feature will relieve the user from having to generate this type of data block. This type of array is commonly used as a basis for customized PRAM stageops and will allow more efficient generation of stagecode.

6.11 STAGEDEFS

An include file of C4PL synonyms for the stage program register offsets and bit offsets into those registers that contain programming switches will be available in C4PL. Descriptive names for the numeric offsets into arrays containing stage program data greatly enhances the readability, debugging and maintenance of procedures which generate stagecode.

6.12 PRAMSET COMMAND

The PramSet command has the form:

PramSet StateIn StateOut Mask InputArray -> OutputArray

The PramSet command is similar to the Cover command, except the input and output variables are arrays. It puts the masked value of StateOut in all locations corresponding to an address of StateIn (under the specified mask).

The purpose of a PRAM stageop is to map a particular input value to a new value. When the stage input and/or output masks are in use, the desired PRAM mapping becomes a little complicated. Masks are typically used to prevent whatever neighborhood operation is programmed in a stage from affecting particular bits of the image.

Suppose we are using the low-order bit of the image being processed to hold some interesting information and we don't want operations on the rest of the bits to affect the low-order bit state. Now suppose we want to map pixel state 2 to state 4. In this situation we would call the PramSet command with StateIn = 2, StateOut = 4, Mask = 254 (0xFE), and the input array is ascending or straight (date=address). PramSet will create an output array based on the input array as follows:

Original:	Modified:
Location 0: 0	0
Location 1: 1	1
Location 2: 2	4
Location 3: 3	5
Location 4: 4	4
Location 5: 5	5
Location 6: 6	6
...	

The C4PL code fragment that implements the PramSet command is a good example of the use of logical operators and array expressions in C4PL.

```
statein := statein & mask
stateout := stateout & mask
bool := (ascending_pram & mask) = statein
outpram := inpram * (~bool) + ( stateout | inpram*(~mask) ) * bool
makearray T_BYTE 256 -> outpram
```

Ascending_pram is a predefined 256-byte array whose data=address. Bool is, therefore, an array of boolean values that defines the addresses of the PRAM that will be modified with the stateout value. The outpram then takes on either the corresponding value from the inpram or a new value that is a combination of bits from stateout and the corresponding inpram value as determined by the specified mask.

6.13 PRAMSWAP COMMAND

PramSwap swaps two values in a PRAM array under control of a bitmask. This command is used when it is desired to exchange two values in an image and is specified as follows:

```
PramSwap State1 State2 Mask InputArray -> OutputArray
```

PramSwap puts the masked value of State1 in all locations corresponding to an address of State2 (under mask). It also puts the masked value of State2 in all locations corresponding to an address of State1 (under mask). PramSwap is implemented in C4PL by a procedure that makes two calls to the PramSet routine.

6.14 STAGEANALYZE COMMAND

This command in C4PL is essential for the programmer who is attempting to generate user-defined stageops. Stageanalyze is a utility that "disassembles" stageops. It provides a formatted output detailing the programming information contained within a stageop or series of stageops.

6.15 OTHER USEFUL COMMANDS

Several other C4PL commands and functions are typically used in the construction of a C4PL procedure. Most of the commands and functions listed below are described in the C4PL User's Manual, Chapter 5 - Procedures [2].

Procedure syntax: Procedure, EndProcedure.

Variables/constants: gdeclare, declare, syn.

Argument checking: findarg_type, type_of, setdef, setret, etc.

Interactive/information: input, pause, wait, print, printl.

Control flow: break, for, if, repeat, while, etc.

Stagecode handling: loadcode, storecode, runout.

One good way to become familiar with C4PL procedure writing is to examine several of the built-in procedures that implement many C4PL commands.

7.0 ARRAYS VERSUS DIRECT STAGECODE REFERENCES

In general, it is somewhat safer and perhaps easier to deal with stage program data as arrays. The question of whether to build and manipulate stage program data as arrays or in the form of stageops is dependent on context, however. Which should be used depends on the data that is to be created, and how it is to be used or modified.

Three reasons exist for operating in the array domain when creating and manipulating stage program data. First, indexing into stageops is more prone to error due to a need to account for the two bytes of header information in each stageop. This means that indicies for stageop data are offset by two from their usual value. The correct index for the desired stageop must also be provided when accessing a stageop contained in a stagecode variable. Second, stageops are often encoded in compacted form through the use of the RAMtype header byte. The stage programmer must take care that each stageop being manipulated is well understood and that the RAM contents of the stageop are known. C4PL will enforce array index limitations on stageops as well as arrays, but this does not guarantee that the programmer knows what kind of stageop is being manipulated. A XFORM-type stageop may contain 256 bytes representing the high NRAM half, but a programmer might erroneously assume that this RAM data represents the low NRAM half (and will never know the difference unless the RAMtype byte is checked, or until erroneous image transforms occur). Third, stageops in C4PL are entities that are utilized through reference, rather than instance. This means that it is possible for multiple stagecode variables to exist that have references to the same stageop. A change to the stageop made via an array reference through one stagecode variable will, therefore, affect the other stagecode. If this linkage between stagecode variables is unknown to the programmer, unexpected results will obviously follow.

After having made a strong case for using arrays to work with stage program data, a qualifier must be inserted. In instances where the stageops to be modified are well understood, and care is taken to index into them correctly; then it is more efficent to access the stageops directly rather than convert them to arrays, make changes, then convert them back to stagecode.

8.0 SOME EXAMPLES

As previously noted, several examples of stagecode generating procedures exist within C4PL itself. Procedures implement many of the EPICAL image processing commands. The C4PL procedure directory on your system is accessible and these procedures can be copied to the user's directory. This is an ideal way to get a running start--use an existing procedure file as a template and modify it as needed. Procedures are stored on VAX/VMS systems in a directory pointed to by the C4PL defined logical name "c4pl$proc". On other systems, the appropriate directory may be deduced by looking at the default command_search search list for any directories that are used to reference "*.def" files.

8.1 MARKTEES

This EPICAL procedure is a simple example of the use of the Match command to build up a more complex image transformation. It does not perform special stagecode manipulations. It is included here as an example to point out that unique or specialized image transformations can be built out of existing C4PL library commands without necessarily having to resort to direct stagecode generation or manipulation. This code is a simplified version of the actual C4PL routine.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;              Environmental Research Institute of Michigan
;                            Copyright - 1990
;
;
;PROCEDURE NAME:     MarkTees
;
;ABSTRACT:        Mark T-connections of lines in state 'FGState'
;
;ENVIRONMENT:     C4PL V2.5
;
;SPECIFICATION:  MarkTees  FGState TeeState Connectivity
;                          InputImage -> OutputImage StageCode
;
;                States 0 through 255 are valid input parameters. For illegal
;                values, an error message will be output to the terminal.
;
;                The input image is assumed to be the binary image of a
;                skeleton in 'FGState' in rectangular or hexagonal coordinates.
;                The output image is a binary image with the centers of the
;                tees in 'TeeState', provided that these centers were in
;                'FGState' in the original image.
;
;DESCRIPTION:   All pixels in state 'FGState' in the input image which
;            .  are the triple-points (or "tees) of FGstate lines are
;               changed to the TeeState.  The connectivity parameter
;               determines the configuration used by this procedure.
;                The word 'configuration' represents:
;                    N-E-S-W               if 'connectivity' is 4 and we're
;                                             in rectangular mode
;                    N-NE-E-SE-S-SW-W-NW   if 'connectivity' is 8 and we're
;                                             in rectangular mode
;                    N-NE-E-SE-S-W         if we're in hexagonal mode
;
;INPUT PARAMS:   FGState:          foreground state         default: 1
;                                   - find tees in this state)
;                TeeState:         output state             default: 2
;                                   - mark tees found by changing
;                                     them to this state)
;                Connectivity:     connectivity             default: 4
;                                   - 4 = 4-way connectivity assumed for
;                                         input image
;                                   - 8 = 8-way connectivity assumed for
;                                         input image
;                                   - illegal values take default
;                InputImage:       input image              default: active
;
;OUTPUT DATA:    OutputImage:      output image             default: active
;                StageCode:        stage code repository    default: default
;                                                             (execute it)
;
;HISTORY:
;       Rev      Date         Author  Description
;       ---      ----         ------  -----------
;       0.0      08 JAN 90    lrt     original code - derived from Pascal
;                                     external task
;       0.1      25 JAN 90    slm     optimized via array expressions
;       0.2      02 FEB 90    lrt     clean up comments; change array name;
;                                     mask contribution bits with input mask
```

```
;       1.3      26 MAR 90     lnt     allow for image to be last input param
;
;       1.5      26 Apr 90     RML     Replaced logic with Match commands
;       x.x      13 Jun 90     dak     simplified for example
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
procedure (FGState, TeeState, Connectivity, InImage)  -> OutImage, StgCode

Declare   im_pos,           &     ; position on image arg in input param list
          im_num,           &     ; number of image type input params
          def_num                 ; number of default input params

syn  min_state         "0"        ; define valid state value range
syn  max_state         "255"

; ****************************** Set defaults for the arguments here.

setdef 1 -> FGState
setdef 2 -> TeeState
setdef 4 -> connectivity
setdef active -> InImage

setret active -> OutImage

; ****************************** Check input parameters.

if ((type_of(FGState) <> T_DEFAULT) & (type_of(FGState) <> T_INT))
   ERROR "*** Foreground state must be an integer."
elseif ((FGState < min_state) | (FGState > max_state))
   ERROR "*** Foreground state must be a value from 0 to 255."
elseif ((type_of(TeeState) <> T_DEFAULT) & (type_of(TeeState) <> T_INT))
   ERROR "*** Output state must be an integer."
elseif ((TeeState < min_state) | (TeeState > max_state))
   ERROR "*** Output state must be a value from 0 to 255."
endif


if ((type_of(connectivity) <> T_DEFAULT) & (type_of(connectivity)<> T_INT))
   ERROR "*** Connectivity must be an integer."
elseif ((type_of(connectivity) = T_INT) & (connectivity <> 4) &
                & (connectivity <> 8))
   connectivity := 4
   printl "*** MarkTees -- Warning: Connectivity has been changed to 4."
endif


; ****************************** Tell user if funny bit masks are used.

if ((INMASK <> 255) | (OUTMSK <> 255))
   printl "*** MarkTees -- Warning: Funny bit mask is in use."
endif


syn fg "FGState"

If HexMode
    Match TeeState      ~fg  FG         &
                       FG  FG ~fg        &
                        ~fg  FG    2       InImage -> OutImage,stgcode
```

```
ElseIf (connectivity=4)    ;rectangular mode

  Match TeeState      #   FG    #    &
                     FG   FG   FG    &
                      #    #    #    4  InImage -> OutImage,stgcode

ElseIf (connectivity=8)
  Match TeeState      #    #   FG    &
                    ~fg   FG  ~fg    &
                     FG  ~fg   FG    8   OR

  Match TeeState    ~fg    #   FG    &
                      #   FG  ~fg    &
                     FG  ~fg   FG    8   OR

  Match TeeState      #  ~fg   FG    &
                      #   FG  ~fg    &
                     FG  ~fg   FG    8   OR


  Match TeeState    ~fg    #   FG    &
                     FG   FG  ~fg    &
                    ~fg    #   FG    8   OR

  Match TeeState      #  ~fg   FG    &
                     FG   FG  ~fg    &
                      #  ~fg   FG    8   OR

  Match TeeState    ~fg    #   FG    &
                     FG   FG  ~fg    &
                      #  ~fg   FG    8   OR

  Match TeeState      #  ~fg   FG    &
                     FG   FG  ~fg    &
                    ~fg    #   FG    8   InImage -> OutImage,stgcode

EndIf   ;HexMode or connectivity 4 or 8

EndProcedure  ;MarkTees
```

8.2 FINDTEES

The EPICAL procedure FindTees is coded as a procedure in C4PL that uses arrays to set up a stage operation that finds T-type intersections of pixels in a certain state (and assuming a certain connectivity). The code presented here is a simplified version of the procedure as it exists in C4PL.

FindTees differs from MarkTees, in that all points except those that meet the "tee" criteria are changed to zero. Marktees only changes the state of pixels that meet its neighborhood criteria--other pixels are unchanged.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;           Environmental Research Institute of Michigan
;
;                       Copyright - 1990
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;MACRO NAME:    FindTees
;
;ABSTRACT:      Find T-connections of lines in state 'FGState' assuming
;               a certain connectivity.
;
;ENVIRONMENT:   C4PL V2.5
;
;SPECIFICATION: FindTees  FGState OutState Connectivity
;                                   InputImage -> OutputImage StageCode
;
;               States 0 through 255 are valid input parameters. For illegal
;               values, an error message will be output to the terminal.
;
;               The input image is assumed to be the binary image of a
;               skeleton in 'FGState' in rectangular or hexagonal coordinates.
;               The output image is a binary image of the center of the tees
;               found in the original image, in OutState, and all other
;               pixels are zero.
;
;DESCRIPTION:   For all pixels in state 'FGState' in the input image, look for
;               3 or more neighbors in state 'FGState' in the configuration
;               given below. Whenever this is the case, change the center
;               pixel's state to 'OutState', otherwise change it to zero.
;
;               The word 'configuration' represents:
;                   N-E-S-W              if 'connectivity' is 4
;                   N-NE-E-SE-S-SW-W-NW  if 'connectivity' is 8 and we're
;                                           in rectangular mode
;                   N-NE-E-SE-S-W        if 'connectivity' is 8 and we're
;                                           in hexagonal mode
;
;INPUT PARAMS:  FGState:          foreground state          default: 1
;                                   - find tees in this state)
;               OutState:         output state              default: 1
;                                   - mark tees found by changing
;                                     them to this state)
;               Connectivity:     connectivity              default: 4
;                                   - 4 = 4-way connectivity assumed for
;                                         input image
;                                   - 8 = 8-way connectivity assumed for
;                                         input image
;                                   - illegal values take default
;               InputImage:       input image               default: active
;
;OUTPUT DATA:   OutputImage:      output image              default: active
;               StageCode:        stage code repository     default: default
;                                                             (execute it)
;
;EXTERNAL:      uses other C4PL commands:  makearray, arraytocode, applycode
;
;I/O & FILES:   none
;
```

```
;HISTORY:
;       Rev       Date          Author  Description
;       ---       ----          ------  -----------
;       0.0       08 JAN 90     lrt     original code - derived from Pascal
;                                       external task
;       0.1       25 JAN 90     dlm     optimized via array expressions
;       0.2       06 FEB 90     lrt     clean up comments; change array name;
;                                       mask contribution bits with input mask
;       0.3       05 MAR 90     lrt     allow for image to be last input param
;       x.x       26 Apr 90     pak     simplified slightly to use as example
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
procedure (FGState, OutState, Connectivity, InImage)   &
               -> OutImage, StgCode

declare  TempCode,          &     ; stage code being built
         control_reg,       &     ; for stagecode control registers
         low_nram,          &     ; first half of NRAM array
         high_nram,         &     ; second half of NRAM array
         hex_mode,          &     ; bit value for HEXMODE
         cnt                      ; for-loop variant

gdeclare zw_bit_xarray            ; for saving bit counts per address

; ****************************** offsets into mode control register

syn  max_mode_off     "0"         ; bit 3 off (do not use maxposition flags)
syn  dyn_bias_off     "0"         ; bit 2 off (no dynamic bias)
syn  out_control      "0"         ; output control bits = "00" (Always NRAM)


; ****************************** offsets into neighbor control registers

syn  disable_off      "0"         ; bit 3 off (disable neighbor switch is off)
syn  disable_on       "8"         ; bit 3 set (disable neighbor switch is on)

; ****************************** offsets into stagecode array

syn  mode_control     "1"         ; control and mask bytes
syn  bias_value       "2"
syn  output_mask      "3"
syn  input_mask       "4"

syn  ne_control       "5"         ; neighbor control bytes
syn  e_control        "6"
syn  se_control       "7"
syn  s_control        "8"
syn  sw_control       "9"
syn  w_control        "10"
syn  nw_control       "11"
syn  n_control        "12"
syn  c_control        "13"

syn  ne_contrib       "14"        ; neighbor contribution bytes
syn  e_contrib        "15"
syn  se_contrib       "16"
syn  s_contrib        "17"
syn  sw_contrib       "18"
syn  w_contrib        "19"
syn  nw_contrib       "20"
```

```
syn  n_contrib          "21"
syn  p_contrib          "22"

; ****************************** constants used in program

syn  array_start        "1"       ; first element in NRam and PRam arrays
syn  array_end          "256"     ; last element in Nram and PRam arrays
syn  NRam_offset        "255"     ; where high NRAM starts relative to full NRam

syn  RegArraySize       "22"      ; number of bytes in stagecode registers
syn  NRamArraySize      "256"     ; number of bytes in half NRam

syn  min_state          "0"       ; define valid state value range
syn  max_state          "255"

syn  equal_zero         "0"       ; test control bits = "00" (Equal Zero)
syn  carry_out          "2"       ; test control bits = "01" (Carry Out)

; ****************************** Set defaults for the arguments here.

setdef 1 -> FGState
setdef 1 -> OutState
setdef 4 -> Connectivity
setdef active -> InImage
setret active -> OutImage

; ****************************** Check input parameters.

if ((type_of(FGState) <> T_DEFAULT) & (type_of(FGState) <> T_INT))
   ERROR "*** Foreground state must be an integer."
elseif ((FGState < min_state) | (FGState > max_state))
   ERROR "*** Foreground state must be a value from 0 to 255."
elseif ((type_of(OutState) <> T_DEFAULT) & (type_of(OutState) <> T_INT))
   ERROR "*** Output state must be an integer."
elseif ((OutState < min_state) | (OutState > max_state))
   ERROR "*** Output state must be a value from 0 to 255."
endif

if ((type_of(connectivity) <> T_DEFAULT) & (type_of(connectivity)<> T_INT))
   ERROR "*** Connectivity must be an integer."
elseif ((type_of(connectivity)=T_INT) & (connectivity<>4) & (connectivity<>8))
   connectivity := 4
   printl "*** FindTees -- Warning: Connectivity has been changed to 4."
endif

; ****************************** If dynamic bias is on, turn off temporarily

if DYNBIA
   printl "*** FindTees -- Warning: Dynamic bias is temporarily being"
   printl "                turned OFF. (for FindTees operation only)."
endif

; ****************************** Tell user if funny bit masks are used.

if ((INMASK <> 255) | (OUTMSK <> 255))
```

```
   printl '*** FindTees -- warning: Tiny bit mask is in use.'
endif


; ******************************** Determine current Hex Mode if connectivity
; ******************************** is 8. Otherwise, Hex Mode is off for conn = 4.

if HEXMODE
   if (connectivity = 8)                  ; want Hex Mode to stay on if conn = 8
      hex_mode := 16                      ; bit 4 set
   else                                   ; connectivity = 4, so Hex Mode is off
      hex_mode := 0                       ; bit 4 off
      printl '*** FindTees -- Warning: Hex Mode is temporarily being"
      printl "                turned OFF. (for FindTees operation only).'
   endif
else                                      ; system Hex Mode is already off
   hex_mode := 0                          ; bit 4 off
endif


; ******************************** create the arrays

makearray  T_BYTE, RegArraySize  -> control_reg
makearray  T_BYTE, NRamArraySize -> low_nram
makearray  T_BYTE, NRamArraySize -> high_nram

; ******************************** Make a bit count array, first time only.
; ******************************** If this array exists globally, do not
; ******************************** execute the code. (Hope that no one else
; ******************************** has changed the values someplace else).

if (type_of(zw_bit_xarray) <> t_array)
  makearray  T_byte, NRamArraySize -> zw_bit_xarray
  for cnt array_start NRamArraySize
    zw_bit_xarray[cnt] := sum(cnt-1)    ; count of bits at each address
  endfor
endif


; ******************************** assign registers in control array

control_reg[mode_control] := hex_mode|max_mode_off|dyn_bias_off|out_control
control_reg[bias_value]   := 0
control_reg[output_mask]  := OUTMSK
control_reg[input_mask]   := INMASK


; **************************** Set up the neighborhood configuration N-E-S-W;
; **************************** enable the others if connectivity is 8. Also
; **************************** set the contribution bits in the same manner.

control_reg[n_control]  :=   disable_off | equal_zero
control_reg[e_control]  :=   disable_off | equal_zero
control_reg[s_control]  :=   disable_off | equal_zero
control_reg[w_control]  :=   disable_off | equal_zero
control_reg[c_control]  :=   disable_off | equal_zero

control_reg[n_contrib]  :=  - ( FGState & INMASK )
control_reg[e_contrib]  :=  - ( FGState & INMASK )
control_reg[s_contrib]  :=  - ( FGState & INMASK )
```

ERIM

```
control_reg[w_contrib]  :=  - ( FGState & INMASK )
control_reg[c_contrib]  :=  - ( FGState & INMASK )

if (connectivity = 4)                        ; connectivity = 4
   control_reg[ne_control] :=   disable_on  | carry_out
   control_reg[se_control] :=   disable_on  | carry_out
   control_reg[nw_control] :=   disable_on  | carry_out
   control_reg[sw_control] :=   disable_on  | carry_out

   control_reg[ne_contrib] :=  0
   control_reg[se_contrib] :=  0
   control_reg[nw_contrib] :=  0
   control_reg[sw_contrib] :=  0

elsaif HEXMODE                                 ; connectivity = 8, hexmode = on
   control_reg[ne_control] :=   disable_off | equal_zero
   control_reg[se_control] :=   disable_off | equal_zero
   control_reg[nw_control] :=   disable_on  | carry_out
   control_reg[sw_control] :=   disable_on  | carry_out

   control_reg[ne_contrib] :=  - ( FGState & INMASK )
   control_reg[se_contrib] :=  - ( FGState & INMASK )
   control_reg[nw_contrib] :=  0
   control_reg[sw_contrib] :=  0

else                                           ; connectivity = 8, hexmode = off
   control_reg[ne_control] :=   disable_off | equal_zero
   control_reg[se_control] :=   disable_off | equal_zero
   control_reg[nw_control] :=   disable_off | equal_zero
   control_reg[sw_control] :=   disable_off | equal_zero

   control_reg[ne_contrib] :=  - ( FGState & INMASK )
   control_reg[se_contrib] :=  - ( FGState & INMASK )
   control_reg[nw_contrib] :=  - ( FGState & INMASK )
   control_reg[sw_contrib] :=  - ( FGState & INMASK )
endif


; ***** Set up NRAM. In the first half (low portion) of the NRam, the center
; ***** fails the test so we want to output a zero. In the second half (high
; ***** portion) the center passes the test so we want to count the neighbors.
; ***** We need to see 3 or more neighbors to be a branch.

low_nram[array_start : array_end] := 0

high_nram[*] := (zw_bit_xarray[*]>=3)*OutState


; ******************************** combine the arrays to form stagecode

arraytocode low_nram, high_nram, control_reg -> ,TempCode


; ******************************** do the operation or store the code generated

if (type_of(StgCode) = T_UNDEFINED)          ;was stgcode specified?
   applycode TempCode InImage -> OutImage     ;no--execute it
elseif (type_of(StgCode) = T_STAGECODE)      ;is stgcode stagecode?
   StgCode := StgCode + TempCode              ;yes--append
else
```

```
   StgCode := TempCode                          ;no--this is first stageop
endif

; ******************************

endprocedure ; FindTees
```

8.3 SKEL4

The Skel4 routine provides an example of utilizing C4PL commands to generate the stagecode desired, then tweeking it slightly to produce interesting variations. Skel4 is currently an unsupported routine available in C4PL. It uses the same Match neighborhood specifications as thc SkelRec4 procedure in C4PL.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;               Environmental Research Institute of Michigan
;
;                          Copyright - 1989
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;MACRO NAME:    skel4
;
;ABSTRACT:      Rectangular skeletonizing operation with 4-way connectivity.
;
;ENVIRONMENT:   C4PL V2.5
;
;SPECIFICATION: skel4  ns fg flesh passes endpts singpts inimage ->
;                                                 outimage, stgcode
;               Perform a skeletonizing operation that maintains 4-way
;               connectivity, with endpoint and single point reduction as
;               options.  A 'normal' skel changes the foreground state
;               to flesh state.  If ns (neighbor state) is specified then a
;               conditional skel is performed, where the foreground is only
;               skeletonized where it is in contact with the ns.  If
;               the foreground (fg) is not specified then all states are
;               skeletonized.
;               Conditional and 'all states' can NOT be combined.
;
;DESCRIPTION:   Reduce foreground state by eating away from four successive
;               directions (E,S,W,N) without breaking 4-way connectivity of
;               foreground area.  Operation must be applied sequentially in th
;               four directions so as not to reduce foreground to nothing in
;               one stage.
;
;INPUT PARAMS:  ns:             neighbor state                default: default
;               fg:             pixel state to skeletonize    default: 1
;               flesh:          new pixel state               default: 0
;               passes:         number of passes              default: 1
;               endpts:         reduce endpoints flag         default: false
;               singpts:        eliminate single points flag  default: false
;               inimage:        input image                   default: active
;
;OUTPUT DATA:   outimage:       output image                  default: active
;               stgcode:        stage code repository         default: default
;                                                               (execute it)
;
;EXTERNAL:      uses other C4PL commands:  match, setdef, setret, rotate_code,
;               type_of
;
;I/O & FILES:   none
;
;HISTORY:
;       Rev     Date            Author  Description
;
;       0.0     29-Mar-89       pak     original code - derived from Pascal
;                                       external task
;       0.1      6-Apr-89       pak     generalized for conditional and all sts
;       0.2     15-May-89       pak     change single pt neighborhood to produce
;                                       results of Pascal skelrec4 exactly
;       0.3     13-Sep-89       pak     some optimizations for speed
;       0.4     06-Dec-89       pak     changes in rotatecode->changes here
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;
procedure (ns,fg,flesh,passes,endpts,singpts,inimage) -> outimage,stgcode

declare tempnex,        &       ; temp storage for hexmode sys variable
        index,          &       ; loop index
        skelcode,       &       ; whole skeleton stage code
        skelcode1,      &       ; first pass stage code
        skelcode2,      &       ; second and subsequent passes stage code
        stageop,        &       ; work area for stageop tweeks for cond. skels
        stg_prefix,     &       ; header pram
        stg_suffix,     &       ; trailing pram
        east,           &       ; test value for match, depends on conditional
        tf                      ; temporary foreground variable

; ****************************** syns for control registers

syn     mode_control            "3"             ;1 +2 (we're indexing stagecode)
  syn           mode_dyn_bias     "4"           ;
syn     ne_contrib              "16"            ;14 +2
syn     e_contrib               "17"            ;15 +2
syn     c_contrib               "24"            ;22 +2

; ****************************** set defaults

; no default for ns (if it's defaulted then we're not conditional)
; no default for fg (if it's defaulted then we're to do all states)
setdef 0 -> flesh               ; skeleton flesh state is 0
setdef 1 -> passes              ; one pass assumed
setdef false -> endpts          ; do not reduce end points
setdef false -> singpts         ; do not remove single points
setdef active -> inimage
setret active -> outimage
;
; ****************************** parameter checking

if ((type_of(ns) <> T_INT) & (type_of(ns) <> T_DEFAULT))
  printl "***skel4 -- Error:  neighbor state not a valid type."
  return
elseif (type_of(ns) = T_INT)
  if ((ns < 0) | (ns > 255))
    printl "***skel4 -- Error:  neighbor state not in 0-255 range."
    return
  endif
endif

if ((type_of(fg) <> T_INT) & (type_of(fg) <> T_DEFAULT))
  printl "***skel4 -- Error:  forground not a valid type."
  return
elseif (type_of(fg) = T_INT)
  if ((fg < 0) | (fg > 255))
    printl "***skel4 -- Error:  forground not in 0-255 range."
    return
  endif
endif

if ((type_of(ns) <> T_DEFAULT) & (type_of(fg) = T_DEFAULT))
  printl "***skel4 -- Error:  can't do a conditional skeleton of all states."
  return
endif
```

```
if (type_of(flesh) <> T_INT)
  printl "***skel4 -- Error:  flesh not an integer."
  return
elseif ((flesh < 0) | (flesn > 255))
  printl "***skel4 -- Error:  flesn not in 0-255 range."
  return
endif

if (type_of(passes) <> T_INT)
  printl "***skel4 -- Error:  passes not an integer."
  return
elseif (passes < 0)
  printl "***skel4 -- Error:  passes is negative."
  return
endif

if (type_of(endpts) <> T_BOOLEAN)
  printl "***skel4 -- Error:  endpts flag not a boolean."
  return
endif

if (type_of(singpts) <> T_BOOLEAN)
  printl "***skel4 -- Error:  singpts flag not a boolean."
  return
endif

temphex := false
if hexmode
  printl "Warning - skel4 temporarily setting hexmode switch off"
  temphex := hexmode
  hexmode := false
endif
;
; ******************************* some initializations
;

empty -> stg_prefix     ; makes these variables into stagecode
empty -> stg_suffix

; ******************************* set temporary foreground variable 'tf'

if (fg<>#)       ; tf exists because of all states option.  Normally tf=fg
  tf := fg
else
  tf := 0        ; could be anything 0-255 (should avoid dummy states)
endif

; ******************************* handle dummy states if flesh state 0
;                                 and set "east" neighbor for match cmds

if (ns<>#)       ; if conditional...
  ; we always use 1 dummy for conditional skeletonizing
  ; we need 2 if the flesh state is 0
  if (flesh = 0)
    exch 0 DUMMY2 -> ,stg_prefix        ; DUMMY2 is temporary flesh state
    cover DUMMY1 DUMMY2 -> ,stg_suffix  ; DUMMY1 maps to (temp) flesh state
    exch 0 DUMMY2 -> ,stg_suffix        ; dummy states go to specified flesh
                                        ;   state (0) when all done
    if (ns = 0)
      ns := DUMMY2                      ; if ns also 0 it must be remapped
    endif
```

```
    else
      cover DUMMY1 flesh -> ,stg_suffix    ; conditional skels need one unused
                                           ;   state for calculations
    endif
    flesh := DUMMY1
    ; in conditional skels the east neighbor is special test case...
    east := ns
  else
    ; not conditional
    if (flesh = 0)
      exch 0 DUMMY1 -> ,stg_prefix         ; a zero output state has special
                                           ;   meaning in NRAM--must change a
                                           ;   flesh state of 0 to something else
                                           ;   during skel processing...
      stg_suffix := stg_prefix             ; and map "something else" back to 0
                                           ;   when done
      flesh := DUMMY1
    endif
    ; non-conditional (and not all states) skels - the east neighbor is normal...
    east := ~tf
  endif

  ;
  ; ****************************** generate skel code using match command

  ; eat one direction at a time (else might eat too much) - East first

  if singpts
    match flesh     # ~tf ~tf &
                  ~tf  tf east &
                  ~tf ~tf ~tf     0 OR

    match flesh     # ~tf  tf &
                  ~tf  tf east &
                   tf ~tf ~tf     0 OR

    match flesh    tf ~tf ~tf &
                  ~tf  tf east &
                   tf ~tf ~tf     0 OR

    match flesh    tf ~tf  tf &
                  ~tf  tf east &
                   tf ~tf  tf     0 OR
  endif

  if endpts
    match flesh     # ~tf   # &
                   tf  tf east &
                    # ~tf   #     0 OR
  endif

  match flesh      tf  tf   # &
                   tf  tf east &
                   tf  tf   #     0 OR

  match flesh      tf  tf   # &
                   tf  tf east &
                    # ~tf   #     0 OR

  match flesh       # ~tf   # &
```

```
                  tf  tf east &
                  tf  tf   #     0        -> ,skelcode1

; ****************************** tweek skelcode1 as needed for variations

if (fg=#)                       ; all state skeletonizing...
  skelcode1[1][mode_control] := skelcode1[1][mode_control] | mode_dyn_bias
  skelcode1[1][ne_contrib:c_contrib] := 0
endif

; ****************************** now generate and append stageops for
;                                         other 3 directions

rotatecode skelcode1  90 hexmode -> ,skelcode1
rotatecode skelcode1 180 hexmode -> ,skelcode1
rotatecode skelcode1 270 hexmode -> ,skelcode1
;
; ****************************** generate 2nd and subsequent passes stagecode

skelcode2 := skelcode1          ; Multiple passes use same code except
if ( (passes>1) & (ns<>#) )     ; in conditional skeletons:
                                ; addl passes need flesh state in the
                                ; east neighbor contribution value.
                                ;
  stageop := skelcode1[1]       ; Extract stageop from stagecode--must do it
  stageop[e_contrib] := -flesh  ; this way to generate a copy of the stage op
  skelcode2[1] := stageop       ; to modify and use in place of original.
  rotatecode skelcode2  90 hexmode -> ,skelcode2
  rotatecode skelcode2 180 hexmode -> ,skelcode2
  rotatecode skelcode2 270 hexmode -> ,skelcode2
endif

; ****************************** multiply by number of passes and add prams

skelcode := stg_prefix + skelcode1 + skelcode2 * (passes-1) + stg_suffix

; ****************************** do the operation or store the code generated

if (type_of(stgcode) = T_UNDEFINED)           ; was stgcode specified?
  applycode skelcode inimage -> outimage      ; no--execute it
elseif (type_of(stgcode) = T_STAGECODE)       ; is stgcode stagecode?
  stgcode := stgcode + skelcode               ; yes--append
else
  stgcode := skelcode                         ; no--make stgcode=skelcode
endif

; ****************************** reset hexmode

hexmode := temphex

; ****************************** we're done!  (with skel4)

endprocedure
```

REFERENCES

1. Stage Programmer's Manual, ERIM Document IPTL-89-294, Environmental Research Institute of Michigan, Ann Arbor, November 1989.

2. C4PL User's Manual, ERIM Document IPTL-88-81, Environmental Research Institute of Michigan, Ann Arbor, October 1989.

BIBLIOGRAPHY

C4PL Advanced Programming Manual, ERIM Document IPTL-88-84, Environmental Research Institute of Michigan, Ann Arbor, June 1987.

Codd, E.F., Cellular Automata, Academic Press, New York, NY, 1968.

Preston, K., and M.J.B. Duff, Modern Cellular Automata, Plenum Press, New York, NY, 1984.

Serra, J., Image Analysis and Mathematical Morphology, Academic Press, New York, NY, 1982.